BRG 19 VIENNA

Vorwissenschaftliche Arbeit

Monolithic vs. Microservices: a comparison based on the frameworks Flask and Django

Verfasst von:	Benedict Armstrong
Klasse:	8A
Schuljahr:	2017/18
Betreuung durch:	Mag. Christian Schöbel

Wien, Februar 2018

Abstract

{Text}

Vorwort

{Text?}

Table of Contents

1 Introduction

http://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/
https://msdn.microsoft.com/en-us/library/ee658098.aspx
http://www.ics.uci.edu/~fielding/pubs/dissertation/introduction.htm

2 Terminology

XML	e**X**tensible **M**arkup **L**anguage
JSON	**J**ava**S**cript **O**bject **N**otation
CRUD	**C**reate **R**etrieve **U**pdate **D**elete
SOA	**S**ervice **O**riented **A**rchitecture
SLA	**S**ervice **L**evel **A**greement
API	**A**pplication **P**rogramming **I**nterface

{Not done}

3 Monolithic Architecture

Traditional software development started off with single executable artefacts, which are now called monoliths. These monolithic applications can be broken down into modules to reduce complexity at programming time but at runtime the run as a single executable artefact, a monolith (Nicola Dragoni, 2017). This style is known as monolithic architecture and is still heavily used in software engineering. The following section will offer a short explanation of the term.

3.1 Definition

Monolithic architecture, in general describes software whose modules cannot be executed seperately (Nicola Dragoni, 2017). As the adjective monolithic, which describes something that is “cast as a single piece” (Merriam Webster, 2017) implies it is all one entity. Similarly, in webdevelopment terms it referes to a software design pattern in which the application is composed all in one piece. The UI is generated as a so called “view” on the server and sent out finished in one piece to the user. Typically, the components are interconnected and again all of these components need to be present for code to be compiled and executed. (Wrox Press Ltd, 1998; Margaret, 2016).

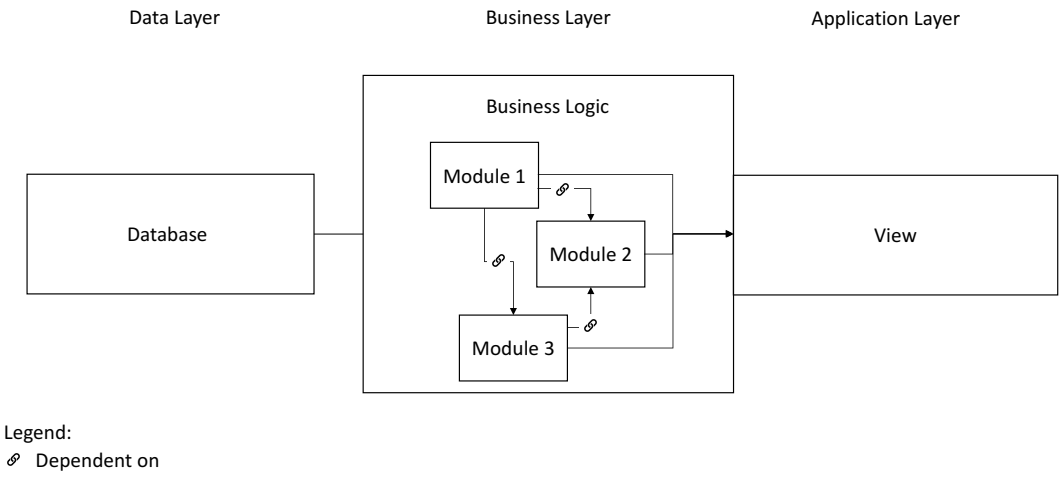

Figure 1, Possible structure of a basic monolithic web application (Annett, 2014)

3.2 Historical Perspective (Nicola Dragoni, 2017)

The development of large-scale applications in the early 60's up to the 70's lead to the first problems and confrontations with software design and it's implications on the development process. The 1970s saw a big increase in references to software design in scientific research and general interest but a solid foundation of the topic was only established by Perry and Wolf in their 1992 book "Foundations for the study of software architecture.". The advent and diffusion of object-orientation, starting in the 1980s and especially in the 1990s, brought its own contribution to the field of Software Architecture. A typical example of an architectural design pattern in object-oriented programming is the Model-View-Controller (MVC), which was used in the development of the cinema seat reservation application §5.4 and is visualized in Figure 1.

4 Microservices Architecture

Microservice architecture is a software design philosophy which describes the logical structure of a software application consisting of microservices.

4.1 Definition of a Microservice

Microservices are modular, loosely coupled components which are lightweight and simple. They should take an input and give back a predictable output. How it calculates the output shouldn't matter, in other words the programming language, the design and the calculations of the service should be replaceable but with a consistent output. (Wolff, 2017)

{needs some reworking}

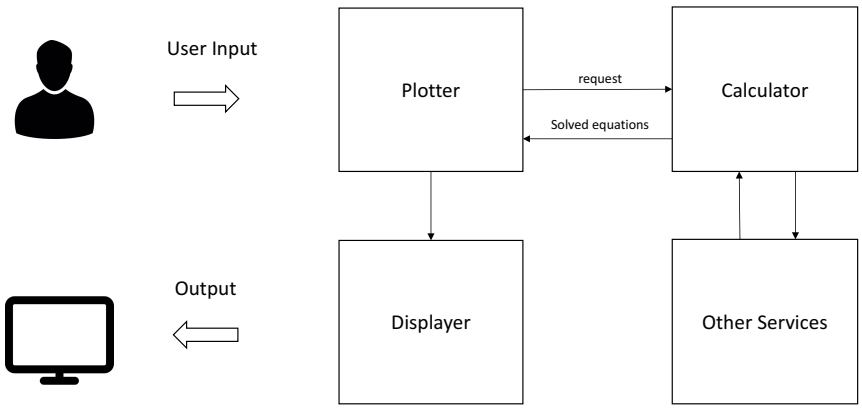

Figure 2, Example of a simple applications consisting of microservices (Nicola Dragoni, 2017)

Consider as an example a microservice which solves equations, so when it receives an input equation it gives back the answers. It should not however do other things like plot the resulting function. That should be handled by a different independent service which can call upon the first service for the calculations. From a web development point of view one of the key differences, compared to a monolithic architecture, is the separation of the presentation layer and the business logic, that means that the UI is composed in the browser on the client's machine instead of the server.

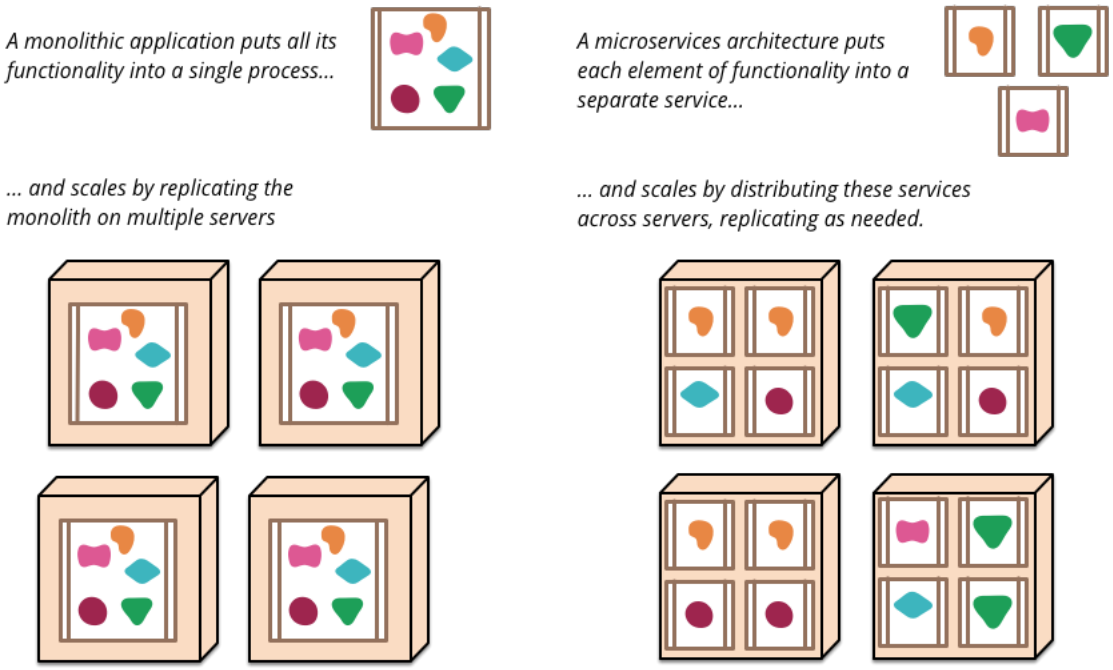

Figure 3, Comparison of Monoliths and Microservices (James Lewis, 2014)

4.2 Historical Perspective

Attention to *separation of concerns* led to Component-based software engineering which has given better control over design, implementation and evolution of software systems. The last decade has seen a further shift towards the concept of services (Service-oriented Computing) first (David Booth, 2004) and the natural evolution to microservices later. (Nicola Dragoni, 2017)
The term "microservices" was first coined at an architectural conference in Venice in 2011 as way to describe the approach of developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a HTTP resource API (James Lewis, 2014). Netflix one of the pioneers of microservice architecture describes its architecture as a "fine grained SOA" (Allen Wang, 2013).

4.2.1 Service-oriented Computing

Service-oriented computing is a paradigm where a program — called a service — offers functionalities to other components, accessible via message passing.

4.2.2 Second generation of services

Service-oriented architecture (SOA) is an architectural style for building software applications that use services available in a network such as the world wide web. It promotes loose coupling between software components, so called services, so that they can be easily reused. A service is an implementation of a well-defined business functionality which can then be consumed by clients in different applications or business processes. (Mahmoud, 2005)
In other words: services decouple their interfaces (i.e. how other services access their functionalities) from their implementation (Nicola Dragoni, 2017).

4.2.3 SOA with Web Services

Web services are software systems designed to support interoperable machine-to-machine interaction over a network (Mahmoud, 2005). A popular approach to this style is REST which I am using to build the cinema seat reservation application §5.3.1.

{Both of these chapters need some restructuring so that everything makes perfect sense}

5 Monolithic and Microservice Architectures in Practice

In order to compare Monolithic and Microservice Architectures I have decided to get some first-hand experience and write my own Application. To highlight the differences, I will go through a call for the movies which are currently showing with each of the two methods.

5.1 Cinema Seat Reservation Application

As a practical real-world example, I have created a web app for reserving seats in a cinema using both approaches. Their user interface is almost identical and kept as simple as possible.

{insert screenshot of UI}

5.1.1 Requirements

The User should be able to pick a Movie from a List and then reserve seats for a specific Time and Date. The System should also support adding and the removal of movies by an administrator. For simplicity, authentication and authorization were ignored.

5.1.2 Structure

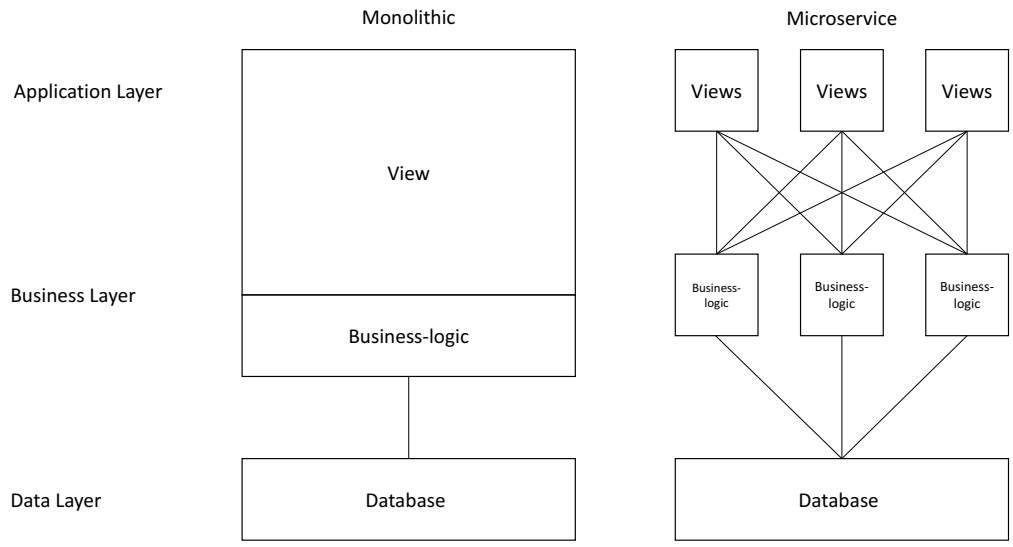

Figure 4, Basic structure of the Cinema Reservation Application (Annett, 2014)

{explain Structure?}

5.1.3 Problems

{Include Section?}

5.1.4 Implementation

5.1.4.1 Database

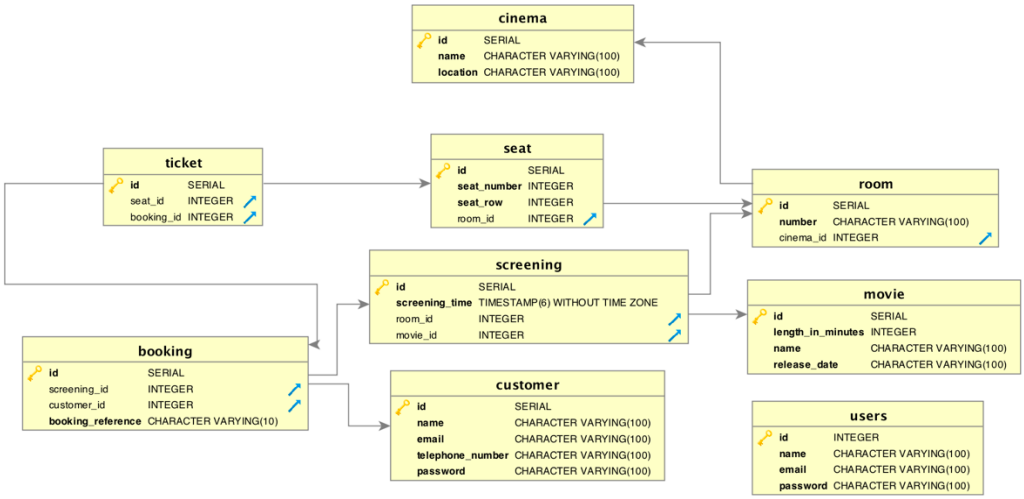

Figure 5, Database Structure

The application uses a Postgres 10 database with a simple, minimalistic layout.

5.2 Get showing movies

To give a better understanding of the application and the structure I will go through an example request, using both approaches, which gets all movies currently showing in a cinema.

5.2.1 Microservices

On the client-side the application uses the JavaScript and the jQuery libraries, the server-side is coded in Python using the Flask and Connexion Frameworks.

5.2.1.1 Client-side

```
1.  $.ajax({
2.      url: "http://127.0.0.1:5000/movies",
3.      success: function(result) {
4.          for (var i = 0; i < result["showing_movies"].length;i++) {
5.              var movie = result["showing_movies"][i];
6.              paint_movie(movie);
7.          };
8.
9.      }, error: function(xhr) {
10.         alert("Error (" + xhr.status + ") :  " + xhr.statusText);
11.     }
12. });
```

The initial request is triggered upon loading of the webpages with the jQuery *ready()* function which executes once the whole document has been loaded in the browser. The *$.ajax()* function shown above is nested within this function. This ajax function performs a HTTP request to the url: *http://127.0.0.1:5000/movies* which is a request to the server, localhost in this case, on port 5000 which the API server is listening on for HTTP requests. The function given as the *success* parameter is executed as the name suggests after a successful request, same with the *error* parameter. In the success function, we process the received *JSON* data, read code below for an example, and for each movie we receive we paint it with the *paint_movie()* function. The error function alerts the user of an error should one occur.

```
{
  "showing_movies": [
    {
      "id": 16,
      "length_in_minutes": 200,
      "name": "Cars",
      "release_date": "2008"
    }
  ]
}
```

5.2.1.2 Server-side

On the server-side the request is first caught by connexion, it filters for the request type and then calls the *showing_movies()* function in controllers/movies.

```
def showing_movies():
    movies = get_movies_showing()
    if (movies is None):
        return "No movies found", 404
    else:
        return {'showing_movies': movies}
```

This takes the return of the *get_movies_showing()* function from the dao and sends it back to Connexion which converts it to JSON and sends it out.

```
def get_movies_showing():
    with get_db_cursor() as cursor:
        cursor.execute("""select * from movie where id in (select movie_id fro
m screening);""")
        return cursor.fetchall()
```

This function performs the actual operation on the database, in this case fetching all the showing movies and returning them.

5.2.2 Monolithic

Here I am using the Django framework and showing a function from the *views.py* module. In this case the routing is handled by a separate module, *urls.py* with regular expressions.

```
def index(request):
    showing_movies_list = (Movie.objects.filter(
        id__in=[s.movie_id for s in Screening.objects.all().distinct()]))
    context = {'showing_movies_list': showing_movies_list}
    return render(request, 'cinema/index.html', context)
```

In the *index()* function, which is one of the views, the showing movies are fetched from the database and passed to the *render()* function, which takes a template, "index.html" in this case, and fills it with the context to render the view.

5.3 Microserviceing

This section will give a brief overview of the technologies used in the application.

5.3.1 Representational State Transfer (REST) **(David Booth, 2004; Fielding, 2000)**

This Application was written using a REST web service architecture. REST is a software architecture defined by Roy Thomas Fielding in his PhD thesis from the year 2000, "Architectural Styles and the Design of Network-based Software Architectures". The REST Web is the subset of the WWW (based on HTTP) in which agents provide uniform interface semantics -- essentially create, retrieve, update and delete -- rather than arbitrary or application-specific interfaces, and manipulate resources only by the exchange of representations. Furthermore, the REST interactions are "stateless" in the sense that the meaning of a message does not depend on the state of the conversation. Objects in the system, called resources, with Uniform Resource Identifiers (URIs) can be manipulated with the operations above, commonly known as "CRUD" operations. REST is often used with the HTTP web protocol and JSON (or XML) as the data format.

5.3.2 Database Operations

These operations are handled by the DAO short for database operations which is written in python and uses the psycopg2 library. First a new connection to the database is opened

```
1.  def get_db_connection():
2.      return psycopg2.connect(
3.        "host=localhost dbname=cinema_reservation_db user=postgres password=***"
4.      )
```

and then a cursor assigned:

```
1.  def get_db_cursor():
2.      conn = get_db_connection()
3.      cursor = conn.cursor(cursor_factory=RealDictCursor)
4.      try:
5.          yield cursor
6.      finally:
7.          conn.commit()
8.          cursor.close()
```

An example function in the DAO might look like this:

```
1.  def get_movie(movie_id):
2.      with get_db_cursor() as cursor:
3.          cursor.execute("""select * from movie where id=%s;""", [movie_id])
4.          return cursor.fetchone()
```

The *get_movie()* function fetches a movie by its id. This function gets called by the controllers.

5.3.3 Controllers

The controllers process the data received from the DAO functions.

```
def info_movie(movie_id):
    movie = get_movie(movie_id)
    if (movie is None):
        return "Movie %s not found" % (movie_id), 404
    else:
        return movie
```

In this example, the info_movie() function calls the DAO function from above and checks weather there is a movie returned and decides what response connexion should send back. A http 404 error, the infamous *"Not found",* is returned if there is no movie with a matching id in the database otherwise it just returns the movie.

5.3.4 Flask

Flask handles the http requests and responses coming from outside. With Flask, it is quite easy to do routing and processing combining the routing and the controller into one function, like in the example below which returns all of the users in the database.

```
@app.app.route('/users/all', methods=['GET'])
def get_users():
    users = get_all_users()
    return jsonify({'users': users})
```

So, with Flask it is possible to leave out the controller or rather compact it but that causes confusion and especially for larger projects I think it is easier to split it up.

5.3.5 Open API with connexion

That's why this application uses connexion on top of Flask. Connexion uses a YAML file which defines the API. In this file, every request is defined with a path, contents and responses. The content of the request can be in the body, in the request header or in the URL or a combination of all. It is good practice to give every request a response, normally a HTTP 200 for a successful request but sometimes something different like a 404 if there is nothing found for a query.

```
/movies:
  get:
    tags: [movies]
    operationId: controllers.movies.showing_movies
    summary: Gets all of the showing movies
    responses:
      200:
        description: Returns showing movies
        schema:
          type: array
          items:
            $ref: '#/definitions/Movie'
      404:
        description: No movies
```

Connexion also uses Open API, previously known as swagger, which gives you a visual representation of the API with handy tools for testing and debugging. This makes it very easy for someone without any knowledge of the underlying business logic to write a client for the application as everything you need to know for a request is precisely detailed in the YAML file and neatly visualised, with examples to try it out.

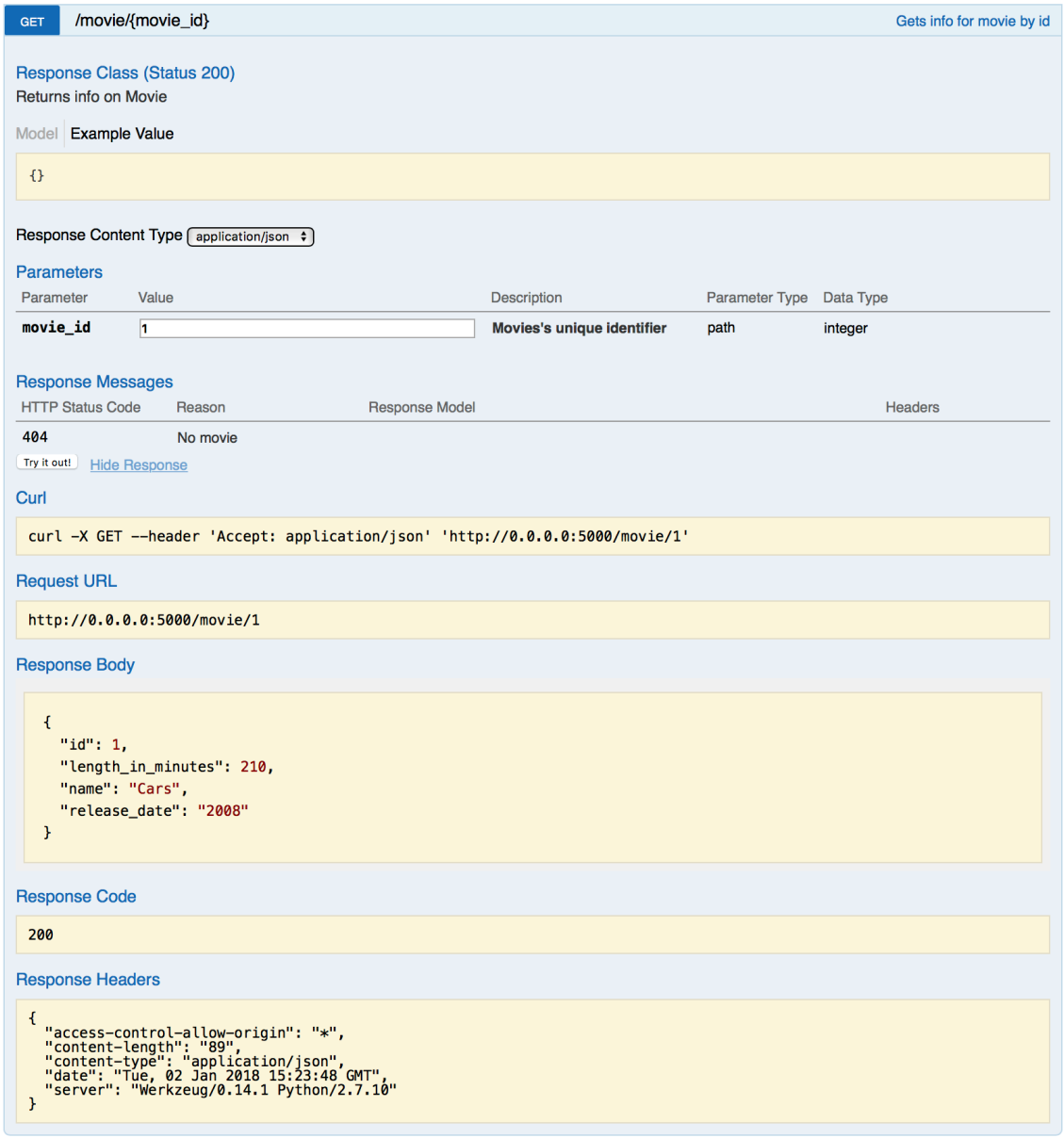

Figure 6, Example of a request detailed in Open API

5.3.6 Client-side

On the client-side requests to the API are made with the jQuery *ajax()* function. It gets a JSON file from the server, converts it to a JavaScript object and passes that to the function in the success parameter.

5.4 Monolithic

A framework is not required but it greatly eases the development so for this paper I have chosen to uses Django, developed and maintained by the Django Software Foundation.

5.4.1 Model-View-Controller

The application is built upon the principles of the Model-View-Controller (MVC) design pattern. “MVC is a pattern used to isolate business logic from the user interface. Using it, the Model represents the information (the data) of the application and the business rules used to manipulate the data, the View corresponds to elements of the user interface such as text, checkbox items, and so forth, and the Controller manages details involving the communication between the model and view. The controller handles user actions such as keystrokes and mouse movements and pipes them into the model or view as required.” (rj45, 2008)

5.4.2 Database Operations with Django Database API

Django provides a neat and easy to use Database API with which SQL statements like this:

```
1.  select * from movie where id in (select movie_id from screening);
```

Can be written in python and the Django Database API. The *QuerySet API* as Django calls it offers useful filters to make complicated SQL statements in python. It also optimizes the number of queries to the database improving performance. (Django Software Foundation, 2018)

```
1.  showing_movies_list = (Movie.objects.filter(id__in=[s.movie_id for s in Screen
    ing.objects.all().distinct()]))
```

5.4.3 Views

{Include this section?}

6 Comparitive Advantages and Disadvantages

This chapter will elaborate on the advantages and disadvantages of either method.

6.1 Monolithic Architecture

There are multiple problems but also some advantages, in no particular order that could make monolithic architecture the right choice for an application:

Advantages:

1. **Simplicity** – Up to a certain size they are in some respects simpler and easier to manage.
2. **Performance –** Monolithic applications’ performance can be higher

Disadvantages:

1. **Learning curve –** Monolithic applications are bound to one language are complex systems are difficult to understand.

2. **Debugging and Tweaking –** Due to tight coupling and the shear complexity of a large monolithic app effective debugging and tweaking can become very difficult.
3. **Teamwork and Development –** Working as a team is made difficult by the language constraint and the single entity nature of a monolithic application.
4. **Reusability –** Parts or modules can only be partially reused as they often have a custom interface and are tightly coupled with the application.
5. **Cross-platform –** Developing cross platform requires a separate server and application.
6. **Scalability –** While not inherently bad monolithic applications are at a disadvantage compared to microservices when it comes to scaling.Introduction

6.1.1 Development

This chapter details the pros and cons of developing a monolithic application. These advantages and disadvantages are based on the Django framework but most of them are relevant to other frameworks.

6.1.1.1 Simplicity

Monolithic architecture is simple for a small system, if everything runs on one server it is easy to deploy and manage. With a simple monolithic application things like eventual consistency and monitoring of the services don't have to be worried about. This is probably the biggest selling point for a monolithic application.

6.1.1.2 Learning curve

Learning monolithic programming might seem simple in the beginning especially with a framework like Django. It is easy to get started with the framework and one can have a functioning web application up and running in no time and without prior knowledge of the framework and concept. This doesn't come without some negative side-effects though, even though you might be able to write a simple application quickly, you'll most likely lack an understanding of the processes going on under the hood. This may eventually lead to problems down the road when you have to scale up the application or customize it further. Django and comparable frameworks are also quite opinionated which can be challenging if you are trying to do things "your own way". The pre-existing code and structure almost forces you, except if you have an excellent knowledge of the framework to adhere to a very strict, predefined way of doing things. This is especially bad for people used to working with other framework which impose different opinions.

An advantage of Django is that a developer can theoretically develop a whole application in one Language. This means it only requires learning one Language to build a simple web app but sadly in practice one can't really get away without JavaScript, HTML, CSS and others.

6.1.1.3 Debugging and Tweaking

Django offers handy tools and helpers for debugging but nevertheless it can quickly become tedious and time consuming due to the structure of the application. Bugs can be hidden multiple layers and dependencies down. To make a new addition or changes the whole system has to be brought down, rebuilt and restarted leading to delays in the development time, especially with bigger systems. Changes propagate through the whole

system and can break otherwise working code. This means systems have to be planned carefully and each change to a module with dependant modules has to be thought through thoroughly checking that each dependant module will still work correctly. This makes it hard to change things after they have been implemented. This is especially bad for upgradability but also maintenance of an application.

6.1.1.4 Teamwork and Development

The more complex a system gets the harder it is for an outsider to understand. Due to the tight coupling and dependencies this can lead to, what in software development is sometimes referred to as, a "big ball of mud", where no single developer or even a group understands the application as a whole (Brian Foote, 1999).
The nature of a monolithic application also requires the developers to use one language for the whole system, causing two further problems. New developers can only be hired if they are familiar with the language or have to learn the language and frameworks/libraries used and secondly a developer cannot choose the language best suited for the task. For example, Django wants the developer to use a slightly altered version of python developed by Django when working with templates and the client-side where JavaScript would usually be used.
Working with a team of developers brings its own set of challenges to the table. Interfaces and dependencies have to be laid out precisely before development to avoid uncertainties and incorrect uses of them. The inherent structure of the application as one entity also means that working on it at the same time requires special tools and thought from the team. Bringing it into production and testing are also affected by this since the whole application has to be brought down to implement a change.

6.1.1.5 Reusability

Due to the lack of a standard interface parts of a monolithic application only offer limited reusability. Modules written in other languages can also not be reused and have to be rewritten. This makes work between teams migrating from existing software or upgrading a lot harder.

6.1.1.6 Cross-platform

In today's fast-moving competitive environment having a mobile application is crucial for any business. Touch and traditional mouse control are very different ways of user interaction with the application and therefore usually require developing separate applications. If a company has an existing monolithic web application there is also no obvious solution for converting it to a mobile application. All of this increases the total cost of ownership and development.

6.1.2 Production

6.1.2.1 Performance

Frameworks like Django usually offer pretty good optimisation and especially for simple applications performance is high. The monolithic nature of the application makes call to separate modules fast as they don't have to go over the network
{more research, paper to back up}

6.1.2.2 Scalability

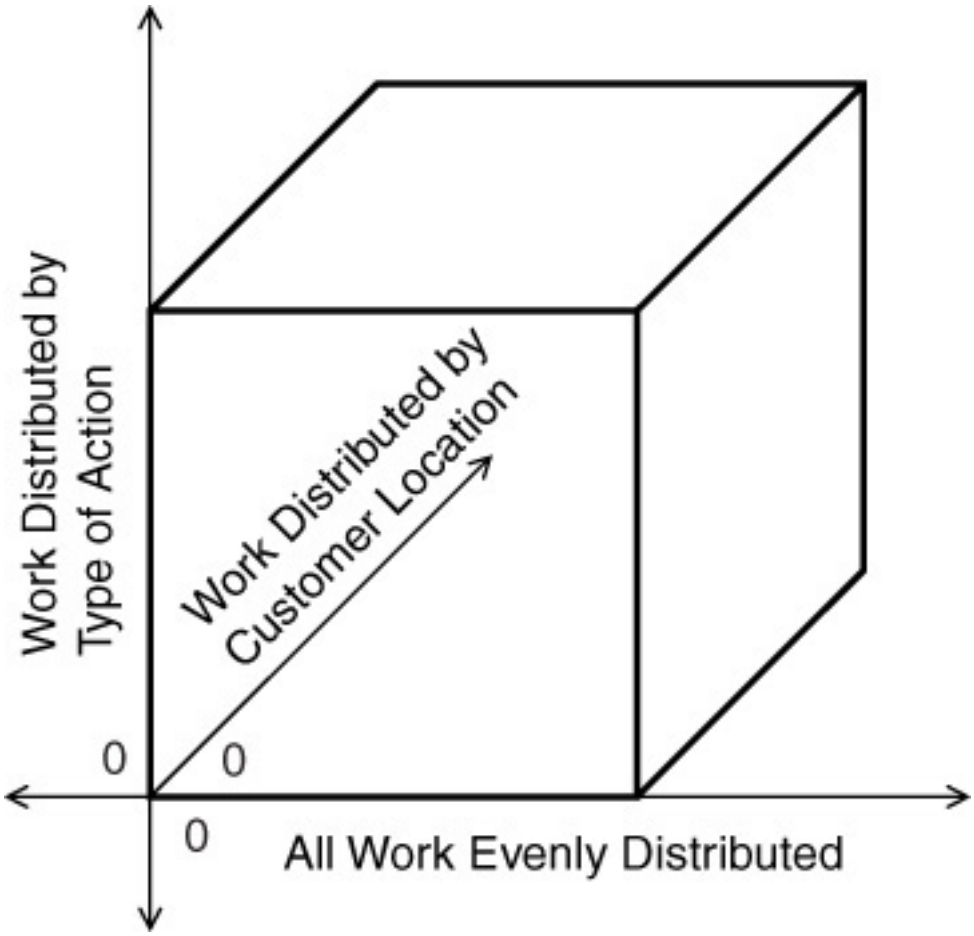

Figure 7, AFK Scale Cube (MARTIN L. ABBOTT, 2015)

Scalability is one of the key factors in application development. Figure 7, is a model to illustrate scalability taken from the book "The Art of Scalability" by Martin L. Abbott and Michael T. Fisher (MARTIN L. ABBOTT, 2015).
The x-axis in this model represents scaling an application by running multiple identical copies of the application behind a load balancer. This of course improves capacity and availability of the system and is probably the simplest way to do so. To optimize this type of scaling it should be noted that the application should be stateless which means each and every request can be routed to most optimal server regardless of previous requests.
The z-axis is scaling by separating and distributing the work by customer, customer need, location, or value this also includes "sharding" in a database which is a way of simplifying and minimizing a database. Unfortunately, this becomes very complicated quickly and makes it very hard to change something in the database. A company could say for example separate the paying Users into a subset with faster or more servers providing higher SLA and the free users into a slower one.

These are the ways of scaling a monolithic application, the third dimension, discussed in the next chapter, makes use of microservices to optimize the scalability to a maximum. So, the y-axis does not apply to monolithic applications which means they provide a lower degree of scalability. It should also be noted that the z-axis is the hardest and most cost intensive way of scaling an application. (MARTIN L. ABBOTT, 2015; Richardson, 2014)

6.2 Microservices Architecture

As with monolithic architecture there are many different frameworks to ease the development process. The application detailed in this paper was written with the Flask framework using the REST approach to micro-servicing. It is what is commonly referred to as a RESTful application.

Advantages:

1. **B2B (Business to Business) integration –** Companies can sell or provide their API to other businesses.
2. **Cross-platform –** Only the client has to be redeveloped for a mobile Application.
3. **Maintainability –** Updating and implementing changes or new features is very simple.
4. **Scalability –** Microservice application can scale in a whole new dimension and naturally have faster and simpler databases
5. **Teamwork and Outsourcing –** Companies can outsource development of a service and coordinating teamwork is easy.
6. **Learning Curve** – Microservices are language independent and the microservice and REST concepts are easy to grasp.

Disadvantages:

1. **Performance –** High latency and network speeds between services can cause performance issues.
2. **Maintainability –** Maintaining a whole network of hundreds of services can be tricky especially with reliability and redundancy take into consideration.
3. **Client Performance –** The clients' machine and browser must have enough processing power to send out many requests simultaneously.
4. **Eventual consistency –** Due to the separation of services information can sometimes be inconsistent in the system for short periods of time

6.2.1 Development

Again, like in the previous chapter, this chapter will detail the pros and cons of using a microservice architecture for a web application. Also like before, these advantages and disadvantages refer specifically to a RESTful application using Flask but are relevant to most other microservice based systems.

6.2.1.1 Learning Curve

Learning to code in a microservice architecture is fairly easy. Services are simple and isolated and can be written in any language. In a well-designed application, a developer can use other services without having to worry about someone else's work. This means that new developers can be hired easily and can start work right away instead of having to get an overview over the entire application first or having to learn a new framework and or language. A microservice based web application is also very similar to SOA which is a very common software development architecture, that many developers are already familiar with. As mentioned above a service is language independent allowing the developer to choose the language best suited for the task at hand.

6.2.1.2 Teamwork and Outsourcing

Working in a team and outsourcing is greatly simplified when developing microservice applications. Due to the loosely coupled nature of services, they can be completely independently updated, changed, taken offline or brought up. Small teams or individual developers can work on separate fully independent services and don't have to know anything about how the other services in the system work. This of course also applies to outsourcing a service to a different company. The only thing everyone has to know about is the Interface, which is standardized and easily documented with interactive tools like Open API, detailed in §5.3.5. This also means that the front and backend can be completely separate and can be worked on by different teams or even companies with little extra effort.

6.2.1.3 B2B (Business to Business) integration

With a microservice based architecture it is easy for companies to provide an API for B2B use. A few prominent examples of this are the google maps API from google or the iframe-API from YouTube for embedding a YouTube video into your application. Companies also have the option to sell their services directly to other businesses.

6.2.1.4 Cross-platform

Developing for multiple different platforms requires little extra effort, at least if it is a well-designed RESTful application. This is because the client calls the services as it needs them and puts it together on the device which entails

6.2.2 Production

6.2.2.1 Performance

The performance of a microservice application is very much dependant of the speed of the network between the microservices. If a microservice has to call 20 other services over a slow connection this will cause a 20-fold delay. The connection speed of the client also matters as a web based microservice application can often require upwards of a hundred calls to build a webpage. This is one of the downsides of microservices the more "fine-grained" the services are the more calls it will require to fetch data from the server. If the user's connection has a high latency, upwards of 100ms for example that means those 100 calls will take quite a while to fulfil, causing a delay and maybe even graphical glitches when building the page in the browser.

6.2.2.2 Client Performance

The client's browser and computer must be powerful enough to handle sending out all of these requests and then processing the information received and painting the webpage. This is more an issue of the past though as computers and browsers have become more than capable of handling such requests. Nevertheless, companies might have to support customers with slow machines and old browsers and therefor this issue needs to be considered.

6.2.2.3 Scalability

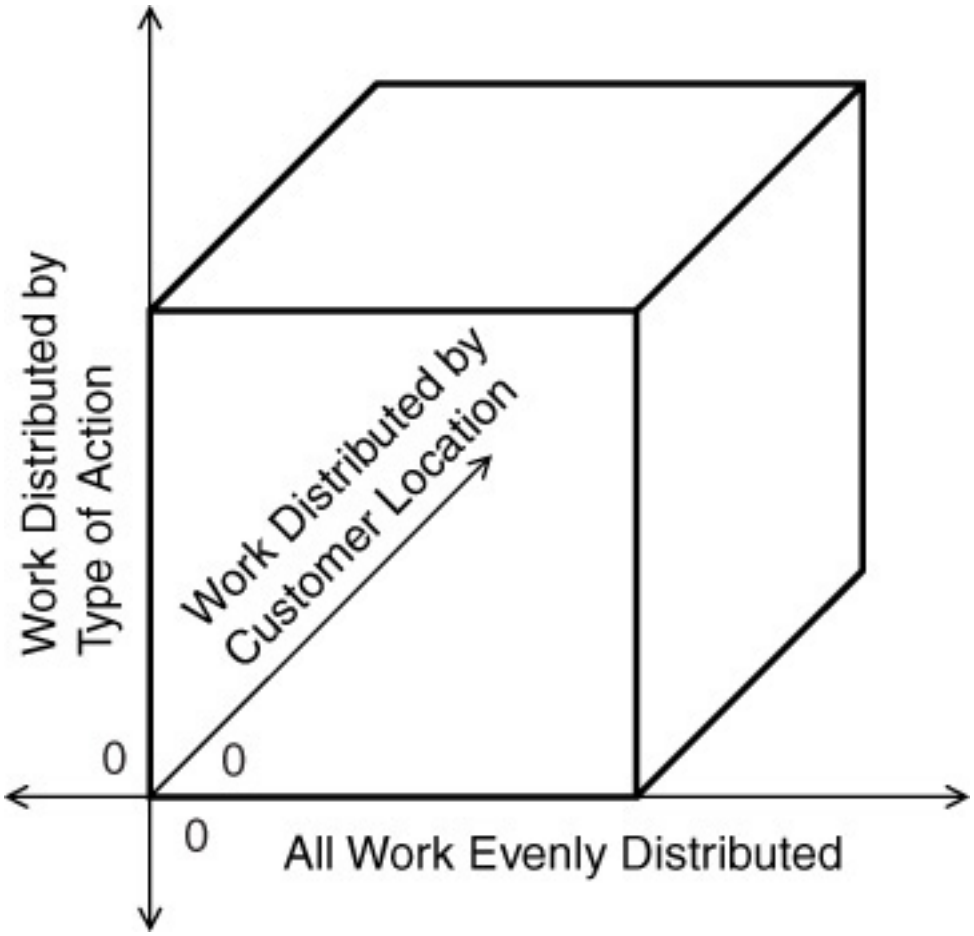

Figure 8, AFK Scale Cube (MARTIN L. ABBOTT, 2015)

Again, we will use the AFK Scale Cube, shown in Figure 8 to visualise scaling. Microservices can of course be scaled along the z-axis by just adding multiple instances of a service running behind a load balancer same concept as with a monolithic application. The z-axis can also be scaled in the same way as with a monolithic application, described in §6.1.2.2. Now comes the new part scaling out into the y-axis, which represents the distribution and separation of work responsibilities or data meaning among multiple entities (MARTIN L. ABBOTT, 2015). In other words, separating the work into microservices.
Due to the independence of microservices it is common practice to use small specific databases for services, this means databases are fast and simple to comprehend.

6.2.2.4 Maintainability

Updating a service is comparatively easy, a microservice can be updated and rewritten as much as needed as long as it still complies with its interface. This means maintaining such a system is in some ways easier because changes and optimisation can be implemented fairly easily.

6.2.2.5 Eventual consistency

If you have different services handling different operations and they don't share a database it can lead to inconsistency "windows" where information is not consistent throughout the whole system yet. This is quite a common problem and often websites will tell you that the update you just made will only show up in a few minutes, in many cases

this is due to this brief inconsistency across the system. This is not just an annoyance for the user though it can cause serious problems for algorithms deciding things based on false or outdated information.

7 Conclusion

In recent years microservice architecture has really taken and is now one of the most hotly debated topic in the web development community. This has led to the adoption of microservice architecture by many companies sometimes without proper consideration of the pros and cons because as laid out in the previous chapter even microservices have their faults.

Simple small-scale applications, like the one written for this paper are still better off as a monolithic application. As a single developer working on an application which won't scale out to support millions of users it is just not worth the effort of maintaining a network of independent services. Changes to a small monolith can be implemented and deployed quickly and such a system is just easier to manage and understand. Simple frameworks like Django which only require one command to build the code and start a server make this especially true. But eventually if the system becomes more and more complicated there comes a point where it is suddenly easier, more productive and in the end, most importantly for a business cheaper to use microservice architecture, as visualized in Figure 9.

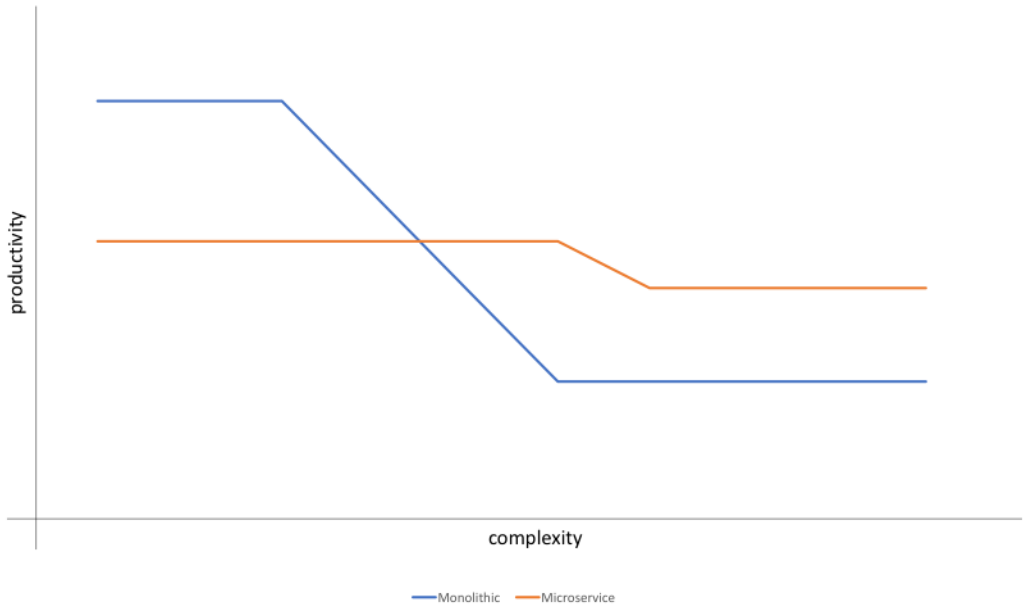

Figure 9, Productivity when developing a web application (Fowler, 2015)

At a certain point with an increasing number of people working on a project and scalability becoming a real constraint microservice architecture clearly takes the upper hand.

{discuss this first}

8 References

Announcing Ribbon: Tying the Netflix Mid-Tier Services Together [Online] / auth. Allen Wang Sudhir Tonse. - 28 January 2013. - 5 January 2018. - https://medium.com/netflix-techblog/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62.

Big Ball of Mud [Online] / auth. Brian Foote Joseph Yoder. - 26 June 1999. - 5 January 2018. - http://www.laputan.org/mud/.

Definition: monolithic architecture [Online] / auth. Margaret Rouse. - 2016. - 14 December 2017. - http://whatis.techtarget.com/definition/monolithic-architecture.

Django [Online] / auth. Django Software Foundation. - 2017. - 15 12 2017. - https://www.djangoproject.com.

Flask [Online] / auth. Opensource / ed. Ronacher Armin. - 2017. - 31 12 2017. - http://flask.pocoo.org.

InfoQ [Online] / auth. Richardson Chris. - 25 May 2014. - 12 January 2018. - https://www.infoq.com/articles/microservices-intro.

Microservice Premium [Online] / auth. Fowler Martin. - 13 May 2015. - 5 January 2018. - https://martinfowler.com/bliki/MicroservicePremium.html.

Microservices Primer [Online] / auth. Wolff Eberhard. - learnpub, 2017. - 3 January 2018. - https://leanpub.com/microservices-primer/read.

Microservices, a definition of this new architectural term [Online] / auth. James Lewis Martin Fowler. - 25 March 2014. - 5 January 2018. - https://martinfowler.com/articles/microservices.html.

Microservices: yesterday, today, and tomorrow [Online] / auth. Nicola Dragoni Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara Fabrizio Montesi, Ruslan Mustafin, Larisa Safina. - 2017. - 3 January 2018. - https://arxiv.org/pdf/1606.04036.pdf.

monolithic [Online] / auth. Merriam Webster. - 2017. - 3 January 2018. - https://www.merriam-webster.com/dictionary/monolithic.

QuerySet API reference [Online] / auth. Django Software Foundation. - Django Software Foundation, 10 January 2018. - 10 January 2018. - https://docs.djangoproject.com/en/2.0/ref/models/querysets/.

Representational State Transfer (REST) [Online] / auth. Fielding Roy Thomas. - 2000. - 5 January 2018. - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI) [Online] / auth. Mahmoud Qusay H.. - April 2005. - 5 January 2018. - http://www.oracle.com/technetwork/articles/javase/soa-142870.html.

Simple Example of MVC (Model View Controller) Design Pattern for Abstraction [Online] / auth. rj45. - 8 April 2008. - 5 January 2018. - https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design.

THE ART OF SCALABILITY [Book] / auth. MARTIN L. ABBOTT MICHAEL T. FISHER. - [s.l.] : Addison-Wesley Professional, 2015.

Three-tier Application Model [Online] / auth. Wrox Press Ltd. - 1998. - 3 Janurary 2018. - https://msdn.microsoft.com/en-us/library/aa480455.aspx.

Web Services Architecture [Online] / auth. David Booth Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion , Chris Ferris , David Orchard. - W3C, Febuary 2004. - 4 January 2018. - https://www.w3.org/TR/ws-arch/#introduction.

What is a Monolith? [Online] / auth. Annett Robert. - 2014. - 3 January 2018. - http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html.

9 Images